Hydra: A Java library for Markov Chain Monte Carlo

Gregory R. Warnes [1]

March 2002

[1]Gregory R. Warnes is a Coordinator, Non-Clinical Statistics, Pfizer Global Research and Development, MS 8260-2104, Eastern Point Road, Groton, CT 06340. E-mail: gregory_r_warnes@groton.pfizer.com. The author would like to thank Adrian E. Raftery, Thomas Lumley, and Anthony Rossini of the Department of Statistics and Department of Biostatistics of the University of Washington for helpful discussions and guidance. This research was supported in part by NIH/NIAID Grant no. 5 T32 AI07450-09, NIH Grant no. 1 PO1 CA76466, NIH Grant no. 1 PO1 CA76466, and ONR Grant no. N00014-96-1-0192.

Abstract

HYDRA is an open-source, platform-neutral library for performing Markov Chain Monte Carlo. It implements the logic of standard MCMC samplers within a framework designed to be easy to use, extend, and integrate with other software tools. In this paper, we describe the problem that motivated our work, outline our goals for the HYDRA project, and describe the current features of the HYDRA library. We then provide a step-by-step example of using HYDRA to simulate from a mixture model drawn from cancer genetics, first using a variable-at-a-time Metropolis sampler and then a Normal Kernel Coupler. We conclude with a discussion of future directions for HYDRA.

Keywords: Markov Chain Monte-Carlo, Gibbs Sampling, Software Library

Contents

List of Tables

List of Figures

1 Introduction

Markov Chain Monte Carlo (MCMC) is a method of performing numerical integration on functions that can expressed as distributions (Metropolis et al., 1953; Hastings, 1970). The strength of MCMC is that it can simulate from distributions without requiring the densities to be properly normalized. This makes it an indispensible tool for Bayesian statistical models, where properly normalizing posterior densities is often impractical or impossible.

After an initial burn-in period, a properly constructed MCMC sampler will generate a sequence of (non-independent) samples, $X_0, X_1, \ldots, X_N$ from a specified probability distribution, Π. Using these samples, the expectations under Π of any function g can be estimated by computing the sample path average $\widehat{\mathbf{E}(g)} = \frac{1}{N}\sum_{t=0}^{N} g(X_t)$.

While most Markov Chain Monte Carlo algorithms are delightfully simple, there are, as of this writing, only two software packages that implement general MCMC algorithms for statistical applications, `WinBUGS` and `FBM`. Both of these have important limitations.

`WinBUGS` (Gilks et al., 1992; Gilks et al., 1994b) is a software package for performing Bayesian inference using Gibbs sampling. It provides tools for specifying the model, running the Gibbs sampler, and monitoring convergence using a "point-and-click" graphical interface. A noteworthy feature is that it allows models to be specified using either a text-based notation or a graphical model created with the `DoodleBUGS` interface (Spiegelhalter et al., 1999).

While `WinBUGS` is mature and is available free of charge from the MRC Biostatistics Unit web site (Stevens, 2000), it has several drawbacks. First, `WinBUGS` is designed to perform only Gibbs and componentwise Metropolis sampling and does not allow specification of alternative sampling methods. As a consequence, `WinBUGS` cannot be used when Gibbs sampling or Metropolis-within-Gibbs are inappropriate. Second, the source code to `WinBUGS` is not available to the user. This makes it impossible for users to add features to `WinBUGS`. Thus, users who require features (such as statistical distributions or sampling methods) not provided by `WinBUGS` are forced to abandon the package entirely. In addition the inability to access the source code prohibits the use of `WinBUGS` for experimentation with or customization of sampling algorithms. This prevents `WinBUGS` from being used as a tool for research on MCMC methods.

Radford Neal's `FBM` ("Flexible Bayesian Modeling") software (Neal, 2000), first released in 1995, is a less well known package that implements a variety of MCMC methods and includes the `C` source code. While `FBM` is more flexible than `WinBUGS`, the `FBM` documentation

and interface are considerably more difficult to understand.

Both `WinBUGS` and `FBM` are restricted to specific operating systems. While older versions of `BUGS` were available for Unix systems, the current version is available only for systems running versions of Microsoft Windows. `FBM`, on the other hand, is available only for Unix systems. In addition, neither package is integrated with standard statistical tools. This requires the user to learn the interface of an additional software package in order to use MCMC.

These drawbacks appear to have discouraged or prevented many users from taking advantage of the considerable effort and expertise represented by `WinBUGS` and `FBM`. As evidence of the general dissatisfaction with the available tools, all of the statisticians we have observed using or researching MCMC write their own custom software. This results in considerable duplication of effort. Worse, since properly debugging and verifying software algorithms is a difficult and time-consuming task, it is likely that many of the hand-written software programs contain undetected errors. This can lead to the presentation of faulty analyses. Finally, lack of integrated software support for MCMC has led many applied researchers to avoid Bayesian statistical methods entirely.

2 Our Approach

Clearly, there is a need for better MCMC software. Our goal is is to produce a software tool that

1. implements standard MCMC techniques,
2. is easy to use,
3. is reliable,
4. is applicable to a wide variety of problems,
5. allows access to the underlying algorithms,
6. can be easily customized and extended,
7. is integrated with traditional statistical packages, and

8. is available on all common computer platforms.

The HYDRA MCMC library is a first step toward providing software that achieves these goals. HYDRA is an object-oriented library that implements the logic of standard MCMC methods. Although HYDRA can be used directly in custom MCMC programs, it is designed to be used as a basis for MCMC software which provides a more user-friendly interface and which is integrated with standard statistical packages.

HYDRA is implemented using Java, a platform independent object-oriented language designed for general programming tasks. We selected Java (Joy et al., 2000) because it enabled the library to meet a number of our stated goals. First, Java's support of formal *interfaces* facilitated the construction of a library that is easy to use without sacrificing flexibility and ease of extension. In particular, the use of interfaces permits users to reploace exisiting components of the MCMC algorithm with versions which are tuned to the specific problem. This allows the user to extend the HYDRA package to support new problems or MCMC techniques without changing the existing code. Second, Java provides features that reduce common programming errors and is supported by a wealth of standard libraries and programming tools. Not only do Java's features make it easier to write code that is error-free, but they also make it easier to locate and correct bugs that do exist. This supports construction of a reliable library and frees time otherwise spent debugging for the development of additional features. Third, Java provides a clear interface for interacting with other languages. This gives a well defined method for HYDRA to be used with existing programming languages and software applications. Although early versions of Java suffered from performance problems, recent versions of the Java virtual machine (runtime) can provide speed comparable to `C` and `C++` for numerical computations (Rijk, 2000; Lewis, 2000; Zachmann, 2000; Schulman, 1997).

The remainder of this text assumes a basic familiarity with the Java language at the level of *Java in a Nutshell*(Flanagan, 1997).

3 Constructing Metropolis-Hastings Samplers Using HYDRA

HYDRA supports the full generality of Markov Chain Monte Carlo by providing a hierarchy of classes that implement the most common MCMC techniques. Classes implement the general Metropolis-Hastings algorithm, the Metropolis sampler, and the Gibbs sampler. HYDRA also implements the multi-state Adaptive Metropolis Sampling (Gilks & Roberts, 1996) algorithm that forms the basis of Adaptive Direction Sampling (Gilks et al., 1994a) and Normal Kernel Coupling (Warnes, 2000). We will focus on the implementation of the Metropolis-Hastings method since it includes the others as special cases.

The Metropolis-Hastings algorithm is remarkably simple. Given a target distribution of interest Π, corresponding to the statistical model, an initial starting location X_0, and a proposal distribution $\mathcal{Q}(X_t)$, each iteration of the sampler consists of four steps:

1. **Propose** a candidate state Y using the proposal distribution $\mathcal{Q}(X_t)$, which may depend on the current state X_t:

$$Y \leftarrow \mathcal{Q}(X_t)$$

2. **Compute** the Metropolis-Hastings acceptance probability

$$\begin{aligned} \alpha(X_t, Y) &= \min\left\{1, \frac{\pi(Y)\ q(Y \to X_t)}{\pi(X_t)\ q(X_t \to Y)}\right\} \\ &= \min\left\{1, \frac{p(Y)\ q(Y \to X_t)}{p(X_t)\ q(X_t \to Y)}\right\} \end{aligned}$$

where π is a density corresponding to the target distribution Π, $p(x) \propto \pi(x)$ is an unnormalized density, and $q(Y \to X_t) = q(Y|X_t)$ is the conditional density of Y under $\mathcal{Q}(X_t)$.

3. **Accept** the proposed point Y and set

$$X_{t+1} \leftarrow Y$$

with probability $\alpha(X_t, Y)$, otherwise

Reject the proposed point and set

$$X_{t+1} \leftarrow X_t.$$

4. **Increment** time: $t \leftarrow t + 1$.

The sequence of X values generated by this algorithm converges to a (dependent) sample from Π provided the proposal distribution Q meets certain conditions (Tierney, 1996).

The `CustomMetropolisHastingsSampler` class implements the logic of Metropolis-Hastings samplers using a target distribution (model), initial state, and proposal distribution specified by the user. This is made possible by requiring the objects representing the target and proposal distributions to provide certain methods. These methods are defined by the `UnnormalizedDensity` and `GeneralProposal` interfaces, respectively. No restriction is placed on the initial state, provided it is compatible with the user-specified target and proposal distributions.

To allow flexible reporting of the progress of the MCMC sampler, the `CustomMetropolis-HastingsSampler` maintains a list of user defined objects that are notified at the completion of the acceptance step of each iteration. When detailed reporting is selected, these "listeners" receive an object containing a great deal of information about each MCMC iteration.

3.1 The `UnnormalizedDensity` Interface for Target Distributions

Target distributions implement the `UnnormalizedDensity` interface, which defines two methods:

```
public double unnormalizedPDF    ( Object state );
public double logUnnormalizedPDF( Object state );
```

These methods compute the (log) unnormalized density of the model for the state passed as a parameter.

3.2 The `GeneralProposal` Interface for Proposal Distributions

Proposal distributions implement the `GeneralProposal` interface, which has 4 methods:

```
public double conditionalPDF    ( Object next, Object current);
public double logConditionalPDF( Object next, Object current );

public double transitionProbability( Object from, Object to );
public double logTransitionProbability( Object from, Object to );
```

The methods `conditionalPDF` and `logConditionalPDF` compute the probability of generating the object `next` when the current state is `current`. The second two methods perform the same computation, but reverse the arguments[1].

3.3 The `MCMCListener` Interface for Listener Objects

Objects that are notified at the completion of each MCMC iteration implement the `MCMCListener` interface, which defines one method:

```
    public void notify( MCMCEvent event );
```

The parameter of the notify method is an object containing information about the MCMC iteration. This information can be used by the object in various ways. Possibilities include storing the current state to a file, displaying it on a plot, and computing cumulative statistics.

When detailed reporting is disabled, the object passed to `notify` is a `GenericChainStepEvent`. This object has a single field:

```
public Object current;
```

which contains the current state (X_t) of the sampler.

When detailed reporting is enabled, the object passed to `notify` is a `DetailChainStepEvent` which has the additional fields:

[1]The `transitionProbability` and `logTransitionProbability` are depreciated and will not be required in a future release of the software.

Table 1: Intepretation of the fields of the `DetailChainStepEvent`

Field	Intepretation
`public Object current;`	X_t
`public Object proposed;`	Y
`public double proposedProb;`	$p(Y)$
`public Object last;`	X_{t-1}
`public double lastProb;`	$p(X_{t-1})$
`public double forwardProb;`	$q(Y\|X_{t-1})$
`public double reverseProb;`	$q(X_{t-1}\|Y)$
`public double probAccept;`	$\alpha(X_{t-1}, Y)$
`public double acceptRand;`	uniform value used to accept/reject
`public boolean accepted;`	was Y accepted?
`public double acceptRate;`	average value of $\alpha(X_{t-1}, Y)$

```
public Object     proposed;
public Object     last;
public double     lastProb;
public double     proposedProb;
public double     forwardProb;
public double     reverseProb;
public double     probAccept;
public double     acceptRand;
public boolean    accepted;
public double     acceptRate;
```

These fields provide a great deal of information about the MCMC iteration and are useful for debugging and for evaluating the performance of different proposal distributions. The interpretation of each field is given in Table 1.

4 Example

The classes provided by HYDRA can be used directly in compiled Java programs or interactively with various Java-based tools, such as `JPython` and the `Omegahat` statistical language. For ease of presentation, we will focus on the pure Java interface.

We will give an example by using HYDRA to construct two different samplers for a Binomial-BetaBinomial mixture model for the loss of genetic material in esopageal can-

Table 2: Class implementing the hierarchical Binomial Beta-Binomial model for the LOH data.

```
1  package org.omegahat.Simulation.MCMC.Examples;
2
3  import java.lang.Math;
4  import org.omegahat.GUtilities.ArrayTools;
5  import org.omegahat.Probability.Distributions.UnnormalizedDensity;
6
7  public class Binomial_BetaBinomial_SimpleLikelihood implements UnnormalizedDensity
8  {
9    int loh[]={ 7, 3, 4, 3, 5, 4, 5, 3, 6, 12, 5, 3, 1, 3, 5, 3, 11, 2, 2, 2, 3, 5, 3,
10             4, 6, 3, 1, 4, 5, 19, 5, 5, 6, 5, 6, 2, 0, 0, 6, 4 };
11   int   n[]={17, 15, 17, 18, 15, 15, 15, 19, 16, 15, 18, 19, 18, 19, 19, 21, 17, 16,
12            12, 17, 18, 18, 19, 19, 15, 12, 16, 19, 16, 19, 21, 15, 13, 20, 16, 17,
13             8,  7, 18, 15};
14
15   // unnormalized binomial density
16   double udb(int x, int n, double pi) {return Math.pow(pi,x)*Math.pow(1.0-pi, n-x);}
17
18   // unnormalized beta-binomial density
19   double udbb(int x, int n, double pi, double omega) {
20     int r; double tmp0 = 1.0; double tmp1 = 1.0; double tmp2 = 1.0;
21
22     for(r=0; r <= (x     - 1 ); r++) tmp0 *= (      pi + ((double) r) * omega);
23     for(r=0; r <= (n - x - 1 ); r++) tmp1 *= (1.0 - pi + ((double) r) * omega);
24     for(r=0; r <= (n     - 1 ); r++) tmp2 *= (1.0 +      ((double) r) * omega);
25     return ( tmp0 * tmp1 / tmp2 );
26   }
27
28   // unnormalized binomial-betabinomial mixture density
29   double ud_b_bb( int x[], int n[], double eta,
30                   double pi0, double pi1, double omega1) {
31     double retval = 1.0;
32
33     for(int i=0; i < x.length; i++)
34       retval *= (      eta) * udb (x[i], n[i], pi0) +
35                 (1.0 - eta) * udbb(x[i], n[i], pi1, omega1) ;
36     return retval;
37   }
38
39   // Constructor //
40   public Binomial_BetaBinomial_SimpleLikelihood() {}
41
42   // Log Unnormalized Density //
43   public double logUnnormalizedPDF( Object parms)  {
44     return Math.log(unnormalizedPDF( parms ));}
45
46   // Unnormalized Density //
47   public double unnormalizedPDF( Object paramObj ) {
48     double[] parms = ArrayTools.Otod( paramObj );
49     double eta=parms[0], pi0=parms[1], pi1=parms[2], omega1=parms[3];
50
51     // check range
52     if( (eta<0.0) || (pi0<0.0) || (pi1<0.0) || (omega1<0.0) ||
53         (eta>1.0) || (pi0>1.0) || (pi1>1.0) || (omega1>0.5) )
54       return 0.0;
55     else
56       return ud_b_bb(loh, n, eta, pi0, pi1, omega1 );
57   }
58 }
```

cers. We first show how to implement the unnormalized density corresponding to Binomial-BetaBinomial mixture model in Java so that it can be used with HYDRA. Using this model we create a variable-at-a-time Metropolis sampler, and then a Normal Kernel Coupler.

4.1 Overview

There are four user-specified components of a Metropolis-Hastings sampler: target distribution (model), initial state, proposal distribution, and uniform random number generator. The HYDRA library provides a reliable random number generator and a selection of standard proposal distributions, leaving the user to construct the target distribution and initial state.

4.2 Creating a Target Distribution

To create an object representing the target distribution (model), the user needs to write a Java class that implements the `UnnormalizedDensity` interface. For our example problem, we wish to implement a class for the Bayesian hierarchical model

$$\begin{aligned} X_i &\sim \eta \text{ Binomial}(N_i, \pi_1) \\ &\quad +(1-\eta) \text{ Beta-Binomial}(N_i, \pi_2, \omega_2) \\ \eta &\sim \text{Unif}(0,1) \\ \pi_1 &\sim \text{Unif}(0,1) \\ \pi_2 &\sim \text{Unif}(0,1) \\ \omega_2 &\sim \text{Unif}(0,1/2) \end{aligned}$$

where the density of the Beta-Binomial distribution is defined as

$$f(X_i|N_i,\pi_2,\omega_2) = \binom{n}{x} \frac{\Gamma(\frac{1}{\omega_2})}{\Gamma(\frac{\pi_2}{\omega_2})\Gamma(\frac{1-\pi_2}{\omega_2})} \frac{\Gamma(x+\frac{\pi_2}{\omega_2})}{\Gamma(n-x+\frac{1-\pi_2}{\omega_2})\Gamma(n+\frac{1}{\omega_2})}$$

A Java class implementing the density for this model is provided in table 2. We shall highlight the programming details that allow use of this class as a target distribution.

First, we need to indicate to the Java compiler where to find the `UnnormalizedDensity` interface that this class will implement. This is accomplished by the line:

```
5 import org.omegahat.Probability.Distributions.UnnormalizedDensity;
```

Now we declare the class and indicate that it implements the `UnnormalizedDensity` interface.

```
7  public class Binomial_BetaBinomial_SimpleLikelihood implements UnnormalizedDensity
```

Next, the class needs a constructor that will accomplish any required initialization, such as loading the observed data. In this case, no initialization is required since the data is hard-coded into the class, so that the line

```
40      public Binomial_BetaBinomial_SimpleLikelihood() {}
```

is sufficient.

Now our class must provide the `unnormalizedPDF` and `logUnnormalizedPDF` methods. These methods are used by the Metropolis-Hastings sampler to compute the acceptance probability for a proposed state.

```
42    // Log Unnormalized Density //
43    public double logUnnormalizedPDF( Object parms)  {
44      return Math.log(unnormalizedPDF( parms ));}
45
46    // Unnormalized Density //
47    public double unnormalizedPDF( Object paramObj ) {
48      double[] parms = ArrayTools.Otod( paramObj );
49      double eta=parms[0], pi0=parms[1], pi1=parms[2], omega1=parms[3];
50
51      // check range
52      if( (eta<0.0) || (pi0<0.0) || (pi1<0.0) || (omega1<0.0) ||
53          (eta>1.0) || (pi0>1.0) || (pi1>1.0) || (omega1>0.5) )
54        return 0.0;
55      else
56        return ud_b_bb(loh, n, eta, pi0, pi1, omega1 );
57    }
```

In this example, we have written separate functions that compute the unnormalized density, so these methods simply convert the arguments to the appropriate type (`doubles`), check their range, call the appropriate function.

Note that the interface defines the argument passed to the `unnormalizedPDF` and `logUnnormalized-PDF` as an `Object`. The user must decide what type of object will represent the model parameters. For most purposes, an array of doubles (`double[]`) is an appropriate choice. For this reason, the predefined proposal methods (see Appendix B) all operate on arrays of doubles. Any other type of object may be used, however, this will require the user to implement an appropriate proposal distribution.

4.3 Creating a Variable-at-a-time Metropolis Sampler

Now that we have a class that implements the unnormalized density for the Binomial-BetaBinomial model, we can construct a MCMC sampler. We first implement a variable-at-a-time Metropolis sampler. The complete class file for this sampler is shown in table 3.

Table 3: Class implementing a variable-at-a-time Metropolis sampler for the LOH model.

```
1  package org.omegahat.Simulation.MCMC.Examples;
2
3  import org.omegahat.Simulation.MCMC.*;
4  import org.omegahat.Simulation.MCMC.Proposals.*;
5  import org.omegahat.Simulation.MCMC.Listeners.*;
6  import org.omegahat.Simulation.RandomGenerators.*;
7  import org.omegahat.Probability.Distributions.*;
8
9  public class Binomial_BetaBinomial_SimpleExample {
10   static public void main( String[] argv ) throws Throwable {
11
12     CollingsPRNGAdministrator a = new CollingsPRNGAdministrator();
13     PRNG prng = new CollingsPRNG( a.registerPRNGState() );
14
15     UnnormalizedDensity target = new Binomial_BetaBinomial_SimpleLikelihood();
16
17     double[] diagVar = new double[]{ 0.083, 0.083, 0.083, 0.042};
18
19     SymmetricProposal proposal =
20       new NormalMetropolisComponentProposal(diagVar, prng );
21
22     double[] state = new double[]{0.90, 0.23, 0.71, 0.49 };
23
24     CustomMetropolisHastingsSampler mcmc =
25        new CustomMetropolisHastingsSampler(state, target, proposal,
26                                            prng, true);
27
28     MCMCListener l = new ListenerPrinter();
29     MCMCListenerHandle lh = mcmc.registerListener(l);
30
31     mcmc.iterate( 10 );
32
33   }
34 }
```

Again we provide the Java compiler with the locations of the classes we will be using. This time there are five `import` statements:

```
import org.omegahat.Simulation.MCMC.*;
import org.omegahat.Simulation.MCMC.Proposals.*;
import org.omegahat.Simulation.MCMC.Listeners.*;
import org.omegahat.Simulation.RandomGenerators.*;
import org.omegahat.Probability.Distributions.*;
```

After declaring the object, we create a `main` function that will do the work of creating and running the MCMC sampler. Within `main`, the first object we need to create is a pseudo-random number generator. The HYDRA library provides an implementation of the Collings random number generator (Collings, 1987), which can be created using the 2 lines:

```
12      CollingsPRNGAdministrator a = new CollingsPRNGAdministrator();
13      PRNG prng = new CollingsPRNG( a.registerPRNGState() );
```

Next, we need to instantiate (create) a copy of our class that implements the unnormalized density of the model. This is accomplished by

```
15      UnnormalizedDensity target = new Binomial_BetaBinomial_Likelihood();
```

Now we instantiate the proposal distribution. For a Metropolis-Hastings sampler, there are several choices (see Appendix B), including a variable-at-a-time random-walk proposal using a normal distribution. This is implemented by the `NormalMetropolisComponentProposal` class. Its constructor allows the specification of a proposal variance for each parameter. We'll use the variance of the parameters under the prior:

```
22      double[] diagVar = new double[]{ 0.083, 0.083, 0.083, 0.042};
23
24      SymmetricProposal proposal =
25        new NormalMetropolisComponentProposal(diagVar, prng );
```

Now we need to define an initial state for the sampler. We'll use the MLE, which is $\eta = 0.90, \pi_1 = 0.23, \pi_2 = 0.71, \omega_2 = 0.49$:

```
22      double[] state = new double[]{0.90, 0.23, 0.71, 0.49 };
```

With the random number generator, initial state, target distribution, and the proposal distribution defined, we can create the actual sampler:

```
24      CustomMetropolisHastingsSampler mcmc =
25         new CustomMetropolisHastingsSampler(state, target, proposal,
26                                             prng, true);
```

This gives us a working MCMC sampler. The final parameter is an optional flag indicating whether the sampler should report all of the details about the MCMC iteration when it calls the listeners, or whether to just report the new state. We wish to see all of the details, so we provide the value `true`.

We need to attach a listener to the MCMC sampler so that we can see the results of each iteration. There is a variety of predefined listeners (see Appendix C), but we'll start with the simplest listener. Its notify method simply displays the object it receives.

```
28      MCMCListener l = new ListenerPrinter();
29      MCMCListenerHandle lh = mcmc.registerListener(l);
```

Finally, with the MCMC sampler defined and a listener attached, we are ready to run the MCMC sampler. This is accomplished by calling the MCMC sampler's `iterate` method with the number of iterations to perform:

```
31      mcmc.iterate( 10 );
```

4.4 Running the Variable-at-a-time Metropolis Sampler

When combined with the HYDRA library, the two classes we've created form a complete Java program. On Unix-like systems with the standard Sun Java tools installed, the classes can be compiled using the `javac` command:

```
> javac Binomial_BetaBinomial_SimpleLikelihood.java
> javac Binomial_BetaBinomial_SimpleExample.java
```

Once the classes are compiled, the MCMC sampler can be run using the Java interpreter by

```
> java org.omegahat.Simulation.MCMC.Examples.Binomial_BetaBinomial_SimpleExample
```

This will cause the MCMC sampler to print detailed information about each of the ten iterations to the screen. The output for the first iteration is:

```
Chain Step Event (with details)
Last            = ContainerState: [ 0.9 0.23 0.71 0.49 ]
Last      Prob  = -359.046964566765
Proposed State  = ContainerState: [ 0.9 0.4751068415506061 0.71 0.49 ]
Proposed Prob   = -423.26454869568283
Current  State  = ContainerState: [ 0.9 0.23 0.71 0.49 ]
Forward Prob    = -0.0369493853787235
Reverse Prob    = -0.0369493853787235
Acceptance Prob = -64.21758412891785
Acceptance Val  = 0.658405257229882
Accepted?       = false
Acceptance Rate = 0.0
```

This gives the current and proposed states, the value of the unnormalized density, the forward and reverse proposal probabilities, the acceptance probability, a flag indicating whether or not the proposed state was accepted, the new state, and the cumulative acceptance rate. Note that the unnormalized density and probabilities are reported on the log scale. The output for all 10 iterations is given in Appendix D

4.5 Enhancing the Variable-at-a-time Metropolis Sampler

This example can be enhanced in a number of ways. First, the class can be modified to use a different proposal method. To use a (complete-state) random-walk Metropolis sampler, simply replace the `NormalMetropolisComponentProposal` with a `NormalMetropolisProposal`. Alternatively, the user could define a custom proposal distribution and use it instead.

Second, it is impractical to store and interpret all of the detailed information produced by using the `StepListenerPrinter` listener for more than a few iterations. Instead, we would like to store just the current state to a disk file. This is accomplished by replacing the `StepListenerPrinter` object with a `StrippedListenerWriter`. Change lines 28 and 29 to

```
28      ListenerWriter l = new StrippedListenerWriter("MCMC.output");
29      MCMCListenerHandle lh = mcmc.registerListener(l);
```

and replace line 32 with

```
32      l.close();
```

The `l.close();` command makes sure that the file that is used to store the MCMC output is properly closed once the MCMC iterations are complete.

Now that the output is being stored to a disk file, it is reasonable to increase the number of iterations. Naturally, this is done by changing the value in the `mcmc.iterate` call to the desired value, say 10,000.

Compiling and running the modified class now generates a data file containing 10,000 MCMC iterations. This output can be read into a standard statistical package for computation of diagnostics and to perform inference. For this, we have found the `CODA`[2] package of MCMC diagnostics, which exists in versions for both `R` and `S-PLUS`, particularly helpful. For either version, the commands

```
library(coda)
mcmc.data <- mcmc(as.matrix(read.table("MCMC.output")));
```

will load the `CODA` library (provided it is installed) and properly import the MCMC data. A selection of diagnostics, plots, and summaries is then available. For instance, the default `CODA` plots and summary statistics for our 10,000 iterations are shown in Figure 1.

[2]See Appendix A.3 for information on obtaining `CODA`.

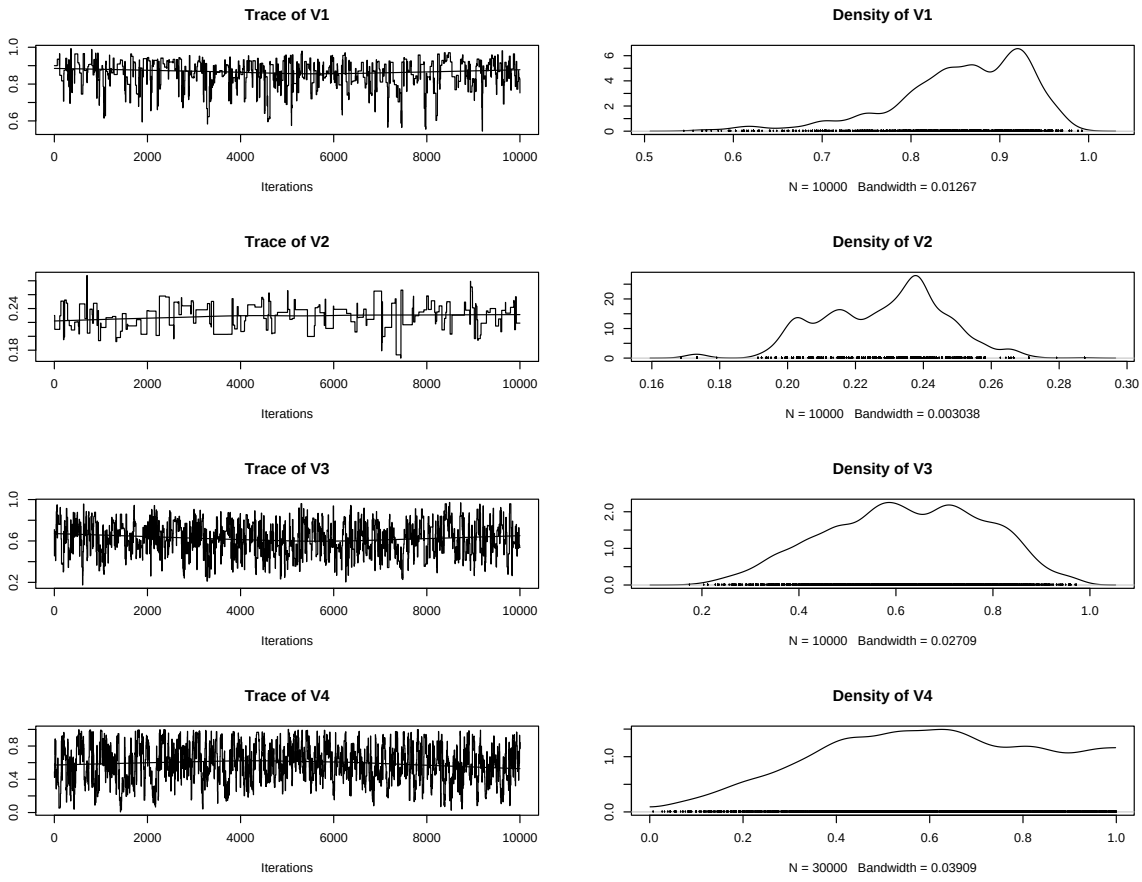

Figure 1: `CODA` plots for 10,000 MCMC iterations.

Table 4: Class implementing a Normal Kernel Coupler for the LOH model.

```
1  package org.omegahat.Simulation.MCMC.Examples;
2
3  import org.omegahat.Simulation.MCMC.*;
4  import org.omegahat.Simulation.MCMC.Proposals.*;
5  import org.omegahat.Simulation.MCMC.Listeners.*;
6  import org.omegahat.Simulation.RandomGenerators.*;
7  import org.omegahat.Probability.Distributions.*;
8
9  public class Binomial_BetaBinomial_SimpleExample_NKC {
10   static public void main( String[] argv ) throws Throwable {
11
12     CollingsPRNGAdministrator a = new CollingsPRNGAdministrator();
13     PRNG prng = new CollingsPRNG( a.registerPRNGState() );
14
15     UnnormalizedDensity target = new Binomial_BetaBinomial_SimpleLikelihood();
16
17     double[][] Var =  {{ 0.003, 0.0  , 0.0  , 0.0  },
18                        { 0.0  , 0.001, 0.0  , 0.0  },
19                        { 0.0  , 0.0  , 0.012, 0.0  },
20                        { 0.0  , 0.0  , 0.0  , 0.007}};
21
22     HastingsCoupledProposal proposal = new NormalKernelProposal(Var, prng );
23
24     int numComponents = 200;
25
26     MultiDoubleState state0 = new MultiDoubleState ( numComponents );
27     for(int i=0; i < numComponents/2; i++)
28         state0.add( new double[]{0.903, 0.228, 0.708, 0.486 } );
29
30     for(int i=numComponents/2; i < numComponents; i++)
31         state0.add( new double[]{0.078, 0.831, 0.230, 4.5e-9 } );
32
33     CustomHastingsCoupledSampler mcmc =
34         mcmc = new CustomHastingsCoupledSampler( state0, numComponents,
35                                                  target, proposal, prng,
36                                                  false );
37
38     ThinningProxyListener pL  =new ThinningProxyListener(numComponents);
39     MCMCListenerHandle    pLh = mcmc.registerListener(pL);
40
41     MCMCListenerWriter l = new StrippedListenerWriter("NKC.output");
42     MCMCListenerHandle lh = pL.registerListener(l1);
43
44     mcmc.iterate( 10000 );
45
46     l.close();
47   }
48 }
```

4.6 Implementing the Normal Kernel Coupler

To implement the Normal Kernel Coupler (NKC) introduced by Warnes (2000), only a three changes need to be made to our example class. First, we use a different proposal distribution. Second, we initialize a set of initial values rather than a single value. Third, we use the `CustomHastingsCoupledSampler` class instead of the `CustomMetropolisHastingsSampler` class. The complete source code for the modified class is given in table 4.

The proposal distribution for the NKC is implemented by the class `NormalKernel-Proposal`. Its constructor requires two arguments, a random number generator, and a matrix that specifies the variance for the normal kernel. For our example, the proposal is instantiated by the lines:

```
17      double[][] Var =  {{ 0.003, 0.0   , 0.0   , 0.0   },
18                         { 0.0   , 0.001, 0.0   , 0.0   },
19                         { 0.0   , 0.0   , 0.012, 0.0   },
20                         { 0.0   , 0.0   , 0.0   , 0.007}};
21
22      HastingsCoupledProposal proposal = new NormalKernelProposal(Var, prng );
```

The NKC maintains a set of current states that must be initialized. We use a `MultiDoubleState`, which holds a list of `double` values, to hold the initial values.

```
24      int numComponents = 200;
25
26      MultiDoubleState state0 = new MultiDoubleState ( numComponents );
27      for(int i=0; i < numComponents/2; i++)
28          state0.add( new double[]{0.903, 0.228, 0.708, 0.486 } );
29
30      for(int i=numComponents/2; i < numComponents; i++)
31          state0.add( new double[]{0.078, 0.831, 0.230, 4.5e-9 } );
```

In this case, we've initialized half of the values to each of the two local maxima.

The logic of multi-state MCMC samplers is implemented by the `CustomHastingsCoupled-Sampler` class. This class is instantiated using 6 parameters, the set of initial states, the number of current states to maintain, the target (model) distribution, the proposal distribution, a random number generator, and a flag indicating whether to report the details of the iteration:

```
33      CustomHastingsCoupledSampler mcmc =
34          mcmc = new CustomHastingsCoupledSampler( state0, numComponents,
35                                                   target, proposal, prng,
36                                                   true);
```

Although we could have simply used a `StrippedListenerWriter` this would generate a very large output file by writing out the entire set of 200 current states at each iteration.

Instead, we use a `ThinningProxyListener` class, which "thins" the events it receives by a specified factor before passing them on:

```
38      ThinningProxyListener pL  = new ThinningProxyListener(numComponents);
39      MCMCListenerHandle    pLh = mcmc.registerListener(pL);
40
41      MCMCListenerWriter l = new StrippedListenerWriter("NKC.output");
42      MCMCListenerHandle lh = pL.registerListener(l1);
```

Running the sampler now will output the complete state once every 200 iterations.

5 Conclusions and Future Directions

Our example has shown that the HYDRA MCMC library makes it easy to create different Metropolis-Hastings samplers without extensive programming. This should encourage additional statisticians to experiment with and use the Metropolis-Hastings method.

We hope that the HYDRA library will form the basis of a set of MCMC tools that are easy to use, robust, and complete. In particular we intend to integrate HYDRA with the statistical tools `R`, `Splus`, and `SAS`, as well as the new `Omegahat` statistical computing language (Temple Lang, 2000; Chambers, 2000; Bates et al., 2000). These interfaces promise to provide flexible and powerful interactive environments for MCMC.

Other goals for the HYDRA library include

- visual tools for specifying and monitoring MCMC simulations
- support for distributed/parallel computing
- a library of target distributions corresponding to common statistical models, such as GLM's and mixture models.

References

Bates, D., Chambers, J., Cook, D., Dalgaard, P., Gentleman, R., Hornik, K., Ihaka, R., Leisch, F., Lumley, T., M achler, M., Masarotto, G., Murrell, P., Narasimhan, B., Ripley, B., Sawitzki, G., Temple Lang, D., Tierney, L., & Venables, B. (2000). The Omega project for statistical computing. web site. `http://www.omegahat.org/`.

Chambers, J. M. (2000). Users, programmers, and statistical software. *Journal of Computational and Graphical Statistics* .

Collings, B. J. (1987). Compound random number generators. *Journal of the American Statistical Association* **82**, 525–527.

Flanagan, D. (1997). *Java in a Nutshell*. O'Reilly & Associates, second edition.

Gilks, W. R. & Roberts, G. O. (1996). Strategies for improving MCMC. In *Markov Chain Monte Carlo in Practice*, pages 89–114. Chapman & Hall.

Gilks, W. R., Roberts, G. O., & George, E. I. (1994a). Adaptive direction sampling. *The Statistician* **43**, 179–189.

Gilks, W. R., Thomas, A., & Spiegelhalter, D. J. (1992). Software for the Gibbs sampler. In *Computing Science and Statistics. Proceedings of the 24rd Symposium on the Interface*, pages 439–448. Interface Foundation of North America (Fairfax Station, VA).

Gilks, W. R., Thomas, A., & Spiegelhalter, D. J. (1994b). A language and program for complex Bayesian modeling. *The Statistician* **43**, 169–177.

Hastings, W. K. (1970). Monte Carlo sampling methods using Markov chains and their applications. *Biometrika* **57**, 97–109.

Joy, B., Steele, G., Gosling, J., & Bracha, G. (2000). *The Java Language Specification*. Addison-Wesley, second edition. also at `http://java.sun.com/docs/books/jls`.

Lewis, J. P. (2000). Java versus C/C++ benchmarks. web site. `http://www.idiom.com/~zilla/Computer/javaCbenchmark.html`.

Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., & Teller, A. H. (1953). Equations of state calculations by fast computing machine. *Journal of Chemical Physics* **21**, 1087–1091.

Neal, R. (2000). Software for flexible Bayesian modeling and Markov chain sampling. Web Site. `http://www.cs.toronto.edu/~radford/fbm.software.html`.

Rijk, C. (2000). Binaries vs byte–codes. *Ace's Hardware (web site)* `http://www.aceshardware.com/Spades/read.php?article_id=153`.

Schulman, A. (1997). Java on the fly. *Webreview.com (web site)* `http://webreview.com/wr/pub/97/07/25/grok/`.

Spiegelhalter, D. J., Thomas, A., & Best, N. G. (1999). *WinBUGS Version 1.2 User Manual*. MRC Biostatistics Unit.

Stevens, A. (2000). The BUGS project. Web Site. `http://www.mrc-bsu.cam.ac.uk/bugs/welcome.shtml`.

Temple Lang, D. (2000). The Omega project: New possibilities for statistical software. *Journal of Computational and Graphical Statistics* .

Tierney, L. (1996). Introduction to general state–space markov chain theory. In *Markov Chain Monte Carlo in Practice*, pages 59–74. Chapman & Hall.

Warnes, G. R. (2000). *The Normal Kernel Coupler: An adaptive Markov Chain Monte Carlo method for efficiently sampling from multi-modal distributions*. PhD thesis, University of Washington.

Zachmann, G. (2000). Java/C++ benchmark. web site. `http://www.igd.fhg.de/~zach/benchmarks/`.

A Installing Hydra

The Hydra Java package is available in two forms, as a Jar file (`Hydra.jar`) containing only the compiled classes and as a gzipped tar file (`Hydra.current.tar.gz`) containing the

full source code as well as the compiled classes. Both files are available from the HYDRA web page located at `http://www.warnes.net/Hydra`.

A.1 Installing the jar File

Download `Hydra.jar` and append the full path to the jar file to the CLASSPATH. For example, if `Hydra.jar` has been placed in the directory `/home/user/jars/`, the proper command for setting the CLASSPATH using `sh` compatible shells is

```
> CLASSPATH=$CLASSPATH:/home/user/jars/Hydra.jar
> export CLASSPATH
```

and using `csh` compatible shells

```
> setenv CLASSPATH $CLASSPATH:/home/user/jars/Hydra.jar
```

A.2 Installing the Full Source

Download `Hydra.current.tar.gz`. It can then be unpacked using GNU tar via

```
> tar -xvzf Hydra.current.tar.gz
```

which will unpack a directory tree with root "Hydra". The Java files and source code are contained in directories under `Hydra/org/omegahat`

The location of this directory then needs to be added to the Java class path. If the directory tree was unpacked in `/home/user/jsrc/` this can be accomplished using `sh` compatible shells by

```
> CLASSPATH=$CLASSPATH:/home/user/jsrc/Hydra
> export CLASSPATH
```

and using `csh` compatible shells

```
> setenv CLASSPATH $CLASSPATH:/home/user/jsrc/Hydra
```

A.3 Other Packages

Two additional Java packages may be required to use particular features of the HYDRA library, Visual Numerics' `JNL` and `Omegahat`. In addition, the `CODA` package, in conjunction

with either R or SPLUS statistical packages, provides a useful suite of tools for evaluating and making inference using MCMC output.

- Visual Numerics' JNL library is required for several of the HYDRA classes, in particular those used in the Binomial-BetaBinomial example given below. JNL is available free of charge from the Visual Numerics web site:
  http://www.vni.com/products/wpd/jnl/.

- The HYDRA MCMC classes were designed to be compatible with the Omegahat statistical programming system. Omegahat provides an interactive environment for statistical programming and analysis and is under active development by the Omegahat project, http://www.omegahat.org.

- The statistical package R is a free re-implementation of the S language and may be obtained free of charge from http://www.r-project.org.

- The CODA package of MCMC diagnostics and other tools for Splus can be obtained from
  http://www.mrc-bsu.cam.ac.uk/bugs/classic/coda04/readme.shtml.

  A version for R can be obtained from http://www-fis.iarc.fr/coda.

B Predefined Proposal Distributions

HYDRA provides a selection of predefined proposal methods for the Metropolis, Metropolis-Hastings, and Hastings-Coupled techniques.

Metropolis Samplers

Class Name	Description
`NormalMetropolisProposal`	normal random-walk proposal
`NormalMetropolisComponentProposal`	variable-at-a-time random-walk proposal

Metropolis-Hastings Samplers

Class Name	Description
`NormalProposal`	(fixed) normal proposal
`NormalMetropolisProposal`	normal random-walk proposal
`NormalMetropolisComponentProposal`	variable-at-a-time random-walk proposal
`MixtureProposal`	finite mixture proposal using specified components

Hastings-Coupled (Multi-Chain) Samplers

Class Name	Description
`IndependentHastingsCoupledProposal`	Wrapper for independent Metropolis-Hastings Samplers
`AdaptiveNormalMetropolisProposal`	(Variance) Adaptive Normal Metropolis Proposal
`AdaptiveNormalProposal`	(Mean, Variance) Adaptive Normal Proposal
`NormalKernelProposal`	Normal Kernel Coupler
`AdaptiveNormalKernelProposal`	(Variance) Adaptive Normal Kernel Coupler
`LocallyAdaptiveNormalKernelProposal`	Locally-Adaptive Normal Kernel Coupler
`KernelDirectionSampler`	Kernel Direction Sampler

C Predefined Listeners

A variety of predefined listeners are available. These allow monitoring various features of the MCMC simulation and give several storage methods.

Class Name	Description
AcceptanceWriter	Stores the cumulative acceptance rate to a file
CovarianceWriter	Stores the cumulative covariance matrix to a file
DistanceListener	Computes the observed and expected acceptance rate, step distance, step distance conditional on acceptance
DistanceWriter	Stores the observed and expected acceptance rate, step distance, step distance conditional on acceptance to a file
HistogramWriter	Stores a cumulative histogram of the current states a file
ListenerGzipWriter	Stores the current state to GZIP compressed file
ListenerPrinter	Prints the event passed to notify()
ListenerWriter	Stores the event passed to notify() to a file
MeanWriter	Stores the cumulative mean vector to a file
PosteriorProbWriter	Stores the (unnormalized) posterior probability of the current state to a file
QuantileWriter	Stores the cumulative quantiles to a file
StepListenerPrinter	Prints `MCMCStepEvents`
StrippedListenerGzipWriter	Stores the current state to a GZIP compressed file
StrippedListenerWriter	Stores the current state to a file
ThinningProxyListener	A proxy for other listeners that thins the reported events by a specified factor, eg 1 out of every 100

D Output from `Binomial_BetaBinomial_Example.java`

```
> java org.omegahat.Simulation.MCMC.Examples.Binomial_BetaBinomial_Example

Chain Step Event (with details)
Last            = ContainerState: [ 0.9 0.23 0.71 0.49 ]
Last     Prob   = -359.046964566765
Proposed State  = ContainerState: [ 0.9 0.4751068415506061 0.71 0.49 ]
Proposed Prob   = -423.26454869568283
Current  State  = ContainerState: [ 0.9 0.23 0.71 0.49 ]
Forward Prob    = -0.0369493853787235
Reverse Prob    = -0.0369493853787235
Acceptance Prob = -64.21758412891785
Acceptance Val  = 0.658405257229882
Accepted?       = false
Acceptance Rate = 0.0

Chain Step Event (with details)
Last            = ContainerState: [ 0.9 0.23 0.71 0.49 ]
Last     Prob   = -359.046964566765
Proposed State  = ContainerState: [ 0.9 0.23 0.45610096830883196 0.49 ]
Proposed Prob   = -360.0165066537818
Current  State  = ContainerState: [ 0.9 0.23 0.45610096830883196 0.49 ]
Forward Prob    = -0.06327351354448152
Reverse Prob    = -0.06327351354448152
Acceptance Prob = -0.969542087016805
Acceptance Val  = 0.31056162077494043
Accepted?       = true
Acceptance Rate = 0.5

Chain Step Event (with details)
Last            = ContainerState: [ 0.9 0.23 0.45610096830883196 0.49 ]
Last     Prob   = -360.0165066537818
Proposed State  = ContainerState: [ 0.9 0.23 0.45610096830883196 0.4225189574152196 ]
Proposed Prob   = -360.1951841070053
Current  State  = ContainerState: [ 0.9 0.23 0.45610096830883196 0.4225189574152196 ]
Forward Prob    = 0.6154440530408976
Reverse Prob    = 0.6154440530408976
Acceptance Prob = -0.17867745322348583
Acceptance Val  = 0.13272539253007873
Accepted?       = true
Acceptance Rate = 0.6666666666666666

Chain Step Event (with details)
Last            = ContainerState: [ 0.9 0.23 0.45610096830883196 0.4225189574152196 ]
Last     Prob   = -360.1951841070053
Proposed State  = ContainerState: [ 1.0245145624463277 0.23 0.45610096830883196
0.4225189574152196 ]
Proposed Prob   = -Infinity
Current  State  = ContainerState: [ 0.9 0.23 0.45610096830883196 0.4225189574152196 ]
Forward Prob    = 0.23049155040119496
Reverse Prob    = 0.23049155040119496
Acceptance Prob = -Infinity
Acceptance Val  = 0.335025050367706
Accepted?       = false
Acceptance Rate = 0.5

Chain Step Event (with details)
Last            = ContainerState: [ 0.9 0.23 0.45610096830883196 0.4225189574152196 ]
```

```
Last      Prob    = -360.1951841070053
Proposed State  = ContainerState: [ 0.9 0.1856011046441249 0.45610096830883196
0.4225189574152196 ]
Proposed Prob   = -363.1065248979661
Current  State  = ContainerState: [ 0.9 0.23 0.45610096830883196 0.4225189574152196 ]
Forward Prob    = 0.3116872397632934
Reverse Prob    = 0.3116872397632934
Acceptance Prob = -2.9113407909608213
Acceptance Val  = 0.14726731467399157
Accepted?       = false
Acceptance Rate = 0.4

Chain Step Event (with details)
Last            = ContainerState: [ 0.9 0.23 0.45610096830883196 0.4225189574152196 ]
Last      Prob    = -360.1951841070053
Proposed State  = ContainerState: [ 0.9 0.23 0.1825256980628881 0.4225189574152196 ]
Proposed Prob   = -364.9924350935826
Current  State  = ContainerState: [ 0.9 0.23 0.45610096830883196 0.4225189574152196 ]
Forward Prob    = -0.1255457772139429
Reverse Prob    = -0.1255457772139429
Acceptance Prob = -4.797250986577353
Acceptance Val  = 0.4310649477090058
Accepted?       = false
Acceptance Rate = 0.3333333333333333

Chain Step Event (with details)
Last            = ContainerState: [ 0.9 0.23 0.45610096830883196 0.4225189574152196 ]
Last      Prob    = -360.1951841070053
Proposed State  = ContainerState: [ 0.9 0.23 0.45610096830883196 0.5879703533000055 ]
Proposed Prob   = -359.85442835072524
Current  State  = ContainerState: [ 0.9 0.23 0.45610096830883196 0.5879703533000055 ]
Forward Prob    = 0.3415983954458079
Reverse Prob    = 0.3415983954458079
Acceptance Prob = 0.0
Acceptance Val  = 0.36058319097411967
Accepted?       = true
Acceptance Rate = 0.42857142857142855

Chain Step Event (with details)
Last            = ContainerState: [ 0.9 0.23 0.45610096830883196 0.5879703533000055 ]
Last      Prob    = -359.85442835072524
Proposed State  = ContainerState: [ 0.6696587069504394 0.23 0.45610096830883196
0.5879703533000055 ]
Proposed Prob   = -361.2666064185844
Current  State  = ContainerState: [ 0.9 0.23 0.45610096830883196 0.5879703533000055 ]
Forward Prob    = 0.00517213125315924
Reverse Prob    = 0.00517213125315924
Acceptance Prob = -1.412178067859145
Acceptance Val  = 0.9268299028867995
Accepted?       = false
Acceptance Rate = 0.375

Chain Step Event (with details)
Last            = ContainerState: [ 0.9 0.23 0.45610096830883196 0.5879703533000055 ]
Last      Prob    = -359.85442835072524
Proposed State  = ContainerState: [ 0.9 0.2099774552506214 0.45610096830883196
0.5879703533000055 ]
Proposed Prob   = -360.42346255905113
Current  State  = ContainerState: [ 0.9 0.2099774552506214 0.45610096830883196
0.5879703533000055 ]
Forward Prob    = 0.32110939780366615
```

```
Reverse Prob     = 0.32110939780366615
Acceptance Prob = -0.5690342083258884
Acceptance Val   = 0.3311132575995816
Accepted?        = true
Acceptance Rate = 0.4444444444444444

Chain Step Event (with details)
Last             = ContainerState: [ 0.9 0.2099774552506214 0.45610096830883196
0.5879703533000055 ]
Last      Prob   = -360.42346255905113
Proposed State   = ContainerState: [ 0.9 0.2099774552506214 0.15833697164158184
0.5879703533000055 ]
Proposed Prob    = -365.44521171685466
Current  State   = ContainerState: [ 0.9 0.2099774552506214 0.45610096830883196
0.5879703533000055 ]
Forward Prob     = -0.2084655958574132
Reverse Prob     = -0.2084655958574132
Acceptance Prob = -5.0217491578035265
Acceptance Val   = 0.7418864833851747
Accepted?        = false
Acceptance Rate = 0.4
```